1987

A System for Experimentation in Replicated Copy Control

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Paul Noll

Andrew Royappa

Donna Sabo

Report Number:
87-725

A SYSTEM FOR EXPERIMENTATION IN REPLICATED COPY CONTROL

Bharat Bhargava
Paul Noll
Andrew Royappa
Donna Sabo

CSD-TR-725
December 1987

A System for Experimentation in Replicated Copy Control *

Bharat Bhargava
Paul Noll
Andrew Royappa
Donna Sabo

Technical Report Number CSD-TR-725

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

This report discusses the implementation of the mini-RAID system used for conducting experiments in replicated copy control during site failure and recovery.

1 Introduction

Designers of distributed database systems must deal with the problem of ensuring the consistency of replicated data during periods of site failure and recovery. A strategy using the concepts of *session vectors* and *fail-locks* has been proposed [1]. To investigate this strategy, the mini-RAID system was developed. Mini-RAID is an abstraction of the experimental system called RAID [2]. The implementation of mini-RAID and the results of experiments conducted with it have been previously described [3]. This report is an extension of that description to provide more details for those interested in the mini-RAID system.

2 Mini-RAID Features

To simulate a distributed system we implemented sites as Unix processes (on one machine). Each process has a copy of the database, fail-locks and session vector and executes the same

*This work was partially supported by grants from UNISYS Corp., NASA, and AIRMICS.

protocol to maintain the consistency of these objects.

We implemented a managing site to provide an interface to the database sites. A current copy of the database and fail-locks is kept by the manager to allow experimenters to dynamically witness changes. This requires the manager to determine which sites are really up and which are really down. A file called *stat.n* exists for each site *n*. Each site keeps its actual session number and state in its file. The manager uses these files to determine which sites are up and which are down. Effectively, these files make up the manager's session vector. The sites do NOT use these files, because each site maintains a local session vector to represent its perception of the status of other sites.

To start the system the manager program, **mgr***, must be executed. It prompts for the following parameters:

1. the maximum total number of read and write operations per transaction. Transactions will be a random number of operations in size within the limits, 1 to the specified maximum. Whether an operation is a read or a write and which database item will be read or written are also randomly chosen

2. the database size in data items

3. the number of database sites (not including the managing site).

The manager forks off the number of sites specified. Before each site execs the **dbsite*** program, it changes its standard output file descriptor to that of a file called *log.n* where *n* is the site number. Each site logs all of the messages that it receives and sends in its own log file. After execing the **dbsite** program, the sites wait for messages to process and the simulation can proceed through the use of the interactive manager program.

An explanation of the manager commands is now given. The help command, **h**, lists the following information:

Simulation commands:
- h = help
- f = fail site
- r = recover site
- x = send user transaction
- m = send multiple user transactions to random sites
- g = send mult. user xacts until fail-locks cleared
- d = cause site to dump information
- o = output current information
- u = output information summary
- c = check on children
- a = send allow recovery
- s = stop simulation

f This command allows the user to fail a particular site. The manager prompts for a destination site and sends a `managing.die` message to that site. Upon receipt of the message the site changes its state to SITE_DOWN and responds to messages from other sites with the `managing.failed` message.

r This command allows the user to bring a site that is down back up. The manager prompts for a destination site and sends that site a `managing.revive` message. Upon receipt of that message, the site executes the protocol described in another section of this report.

x This command allows the user to send a randomly generated user transaction to a particular site. The manager prompts for the site number and then sends a `xact.user` message to the site if the site is up. After receiving the message the site executes the protocol described in another section of this report.

m This command allows the user to send multiple user transactions to sites at random. The manager prompts for the total number of transactions to send. The manager then sends randomly generated transactions in a serial manner to randomly chosen operational sites.

g This command allows the user to send multiple user transactions to sites at random until all of the fail-locks for some site with fail-locks set are cleared. This command is useful when studying the number of transactions it takes to clear fail-locks. The manager sends randomly generated transactions to random sites serially until all of the fail-locks are cleared for a site.

d This command is used to tell a particular site to write its session vector, database and fail-locks to the *log.n* file. The manager prompts for the destination site and sends a `managing.dump` message to the site. Regardless of what state the site is in, the site dumps the information out to its *log.n* file.

o This command is used to cause the manager print its copy of the session vector, database and fail-locks.

u This command is used to print a shorter version of the managing site's data.

c This command is used to check the status of all processes. The UNIX command *ps* is exec-ed.

a This command is used to cause a site to send a recovery response message to another site. This feature exists to allow the simulation of the window of vulnerability which

exists in the time between a site sending a `recovery.announce` message and the site receiving the `recovery.response` message. During this time all other sites will think that the recovering site is up but the recovering site won't really be up until it receives and processes the `recovery.response` message (which contains the fail-locks and new session vector). Unfortunately, there was insufficient time to fully implement this feature. However, the work to be done in order to implement the window is outlined in the discussion portion of this report.

s This command is used to stop the simulation. The manager sends `managing.stop` messages to all sites and then exits. After receiving the `managing.stop` message each site exits.

3 Theoretical Data Availability

This section contains an analysis of the theoretical data availability for transaction processing with the mini-RAID system.

The fact that transactions are randomly generated in our implementation may be of concern to some who say that in reality transactions are not random and actually all data items are probably accessed with different probabilities. We are making the assumption that in the database there is a subset of data items that are frequently referenced. We also assume that the objects in this set have approximately equal probabilities of being accessed. The remaining part of the database is accessed less frequently. The inclusion of rarely accessed data items in our experiments would not significantly alter our results.

Another issue that we must address is the fact that studies have shown that typically reads are far more common than writes. We have implemented the number of reads to be approximately equal to the number of writes. This assumption may be to our disadvantage during the time that fail-locks are being set. A fail lock is set for each down site every time a write operation is performed on a data item. This reduces our data availability more quickly than if we had assumed that writes occur less often than reads. However, this assumption also has the effect of increasing data availability more quickly during recovery with fewer copier transactions. In our implementation many of the fail-locks were cleared by writes instead of by copier transactions requested by a recovering site. If reads occur more commonly than writes then more copier transactions would probably be requested by a recovering site during recovery.

Since transactions are randomly and independently generated in this implementation, we can calculate the average number of write operations in a transaction.

Let M be the maximum number of operations (read or write) in a transaction. Since the actual transaction size is chosen randomly from the set $\{1, 2, \ldots, M\}$, each possible transaction size is chosen with equal probability: $P_t = 1/M$. Furthermore, the average transaction size μ_t is equal to $(M + 1)/2$.

A transaction consists of operations that can be any combination of reads and writes. Since the operation type is randomly chosen from the set $\{read, write\}$, the probability that the operation is a read, P_r, is the same as the probability that the operation is a write, P_w, which is equal to 1/2.

The average number of write operations per transaction is equal to the average number of operations in a transaction multiplied by the probability that a transaction is a write. Since $P_w = P_r$, we have that $\mu_r = \mu_w = (U + 1)/4$.

From this analysis we conclude that our use of completely random transactions did not affect our observed recovery rates but may have affected the observed number of copier transactions requested. We would expect an increased number of copier transactions for systems which have more read operations than write operations.

4 Communications

As stated earlier, a database site is simulated by a UNIX process. Each site is identified by a unique positive integer, called the site ID (SID). Communication between sites is performed by two primitives, `Send` and `Receive`.

All messages in this system are uninterpreted 512 byte buffers. Messages are unreliable, i.e. delivery is not guaranteed. A message can be sent to site N by calling `Send` with two parameters, a string and the integer SID of the destination site. A site can receive messages by calling `Receive`, which will return via parameters a string and the integer SID of the site which sent the message. `Receive` blocks indefinitely until a message arrives; if necessary a timeout mechanism will be added to make it return if no message arrives in certain interval.

Here are descriptions of the most important communication routines:

```
Setup(SID)
int SID;
{
    /*
    ** Must be called with the site ID of this site, and this must
    ** happen before any other communications routine is invoked.
    **
    ** Returns ERROR if something goes wrong.
    */
}

extern int MySID; /* After calling Setup, the SID of this site.*/

Send(buf, SID)
char *buf;
```

```
int SID;
{
    /*
    ** Sends 512 bytes of information starting from buf to the
    ** site whose id is SID.
    **
    ** Returns ERROR if the site is not up, or if something
    ** goes wrong; returns OK otherwise.
    */
}

Receive(buf, pSID)
char *buf;
int *pSID;
{
    /*
    ** Receive will block until a message from another site
    ** appears, or a timeout occurs. The pointer buf must
    ** point to at least MAX_MSG_LEN bytes. The source site's
    ** SID will be filled into the integer pointed to by pSID.
    **
    ** If no message arrives within a certain number of seconds
    ** (currently 30), Receive will return the value TIMEOUT.
    ** In this case the contents of the data area and the
    ** SID integer are undefined (even if you assigned
    ** to them before the call).
    **
    ** If an error occurs, ERROR will be returned; returns OK otherwise.
    */
}

ExitSite()
{
    /*
    ** This must be called when a site exits so that
    ** cleaning up actions can take place. Once a site
    ** calls ExitSite it won't be able to receive any
    ** messages (unless some were already "in transit").
    **
    ** Returns ERROR if anything goes wrong; returns OK otherwise.
```

```
    ** If this returns ERROR, you're probably hosed bad ..
    */
}
```

5 Implementation of Session Vector and Fail-Locks

The session vector is implemented as an array of records, with each record representing a site. The information maintained for a site includes the perception of the site's session number and the site's state. The state field is kept independent of the session number to provide the needed flexibility for the case of recovery following multiple site failures.

Fail-locks are implemented such that each data-item has an independent bit map. The size of each bit map is less than or equal to the number of possible sites. Each bit represents a site with a value of 1 in the *n*th bit indicating that a fail-lock is set for the *n*th site for the data-item. This implementation allowed the fail-lock operations to be performed very quickly, although it may not be suitable for use with systems which have a large number of data-items.

6 Message Definition

Messages are classified by a type and subtype. The message types supported include:

- **control**: these messages are associated with site transitions. Subtypes include:

  recovery_announce used by a site to announce that it is prepared to become operational (corresponds to a type-1 control transaction)

  recovery_response used by operational site to send a session vector and fail-locks to a recovering site

  recovery_wait sent by a recovering site to another recovering site if both sites must await the recovery of the last site to fail

  failure_announce sent by an operational site to other sites after detection of site failure (corresponds to a type-2 control transaction)

  clear_fail_locks sent to clear fail-locks after a copier transaction

  status sent by the last site down after all sites have failed to determine which sites are awaiting its recovery

- **xact**: these messages are associated with database transactions. Subtypes include:

  user a database transaction initiated by a user

update a transaction to write values from the user xact to the database of another site

ack sent by a site to acknowledge reception of `xact.update`

commit sent to a site to indicate that the values of a `xact.update` can be committed

commit_ack sent by a site to acknowledge commitment of `xact.update` values

copier a transaction to read values from the database of another site

copier_update a transaction to return requested copier values

- **managing**: these messages are used to manage the system. Subtypes include:

stop cause a site to gracefully terminate after completion of the simulation

revive cause a site that is down to start recovering

die cause a site that is up to go into failure mode

dump cause a site to dump its current information into its log file

up used in response to a `control.status` message to indicate that the site is up

failed used by a site which is in failure mode to indicate to another site that it is not functioning (instead of having the operational site time out)

xact_committed sent to the manager to indicate that a `xact.user` was committed

xact_aborted sent to the manager to indicate that a `xact.user` was aborted

allow_recovery used to simulate the window of vulnerability between the time that a site sends out a `control.recovery_announce` and the time it receives the `control.recovery_response`. When a site receives this message it will send the `control.recovery_response` to the specified site.

7 Protocol Definition

As stated, each site initially enters a state in which it is waiting for a message of any type. Depending on the message received, the site may begin or take part in a relatively long but well-defined protocol. Once the protocol is finished, the site reenters this initial state and waits for any message. This section describes what each site does upon receipt of a message when it is in this initial state. The protocols can be easily followed with this description. Two conventions that are used in this section are as follows: Receive(**type.subtype**) indicates that a site has received this message and Send(**type.subtype**) indicates that a site sends this message.

Send(**managing.stop**)
simulation over => site should gracefully exit

```
Send(managing.revive)
    if site is failed then
        make a copy of the session vector, oldsv;
        indicate in new session vector, newsv, that
            status of other sites is unknown;
        look at oldsv to see if this site was the
            last one to fail;
        if this was the last site to fail then
            for each other site
                Send(control.status) ;
                get a reply;
                case
                    Receive(managing.failed) :
                        mark site down in newsv;
                    Receive(control.recovery_announce) :
                        set site session number in newsv;
                    Receive(other) :
                            ERROR
                endcase
            endfor
            Send(control.recovery_response)  to all other
                recovering sites with newsv and fail-locks;
        else not the last site down
            for each other site
                Send(control.recovery_announce) ;
                get a reply;
                case
                    Receive(control.recovery_response) :
                        get session vector and fail-locks
                            from msg and become operational;
                    Receive(control.recovery_wait) :
                        last site down is not up yet so mark
                            newsv to indicate that this site
                            this site is waiting;
                    Receive(control.recovery_announce) :
                        increment counter and update newsv;
                    Receive(managing.failed) :
                        increment counter and update newsv;
                endcase
```

```
            endfor
            if received all control.recovery_announce
                and managing.failed then
                all sites went down and some are trying
                    to recover;
                if all other sites in the oldsv which
                    were marked up when this site went
                    down have sent recovery_announce then
                    there were simultaneous site failures
                    if I am the highest ranked site then
                        indicate in newsv that no sites
                            are in recovery wait;
                        mark self as up in newsv;
                        Send(control.recovery_response)
                            to all sites with newsv and fail-locks;
                    else
                        ERROR since the highest ranked site should
                            have sent the control.recovery_response;
            else wait on recovery
                Send(control.recovery_wait) to all
                    sites trying to recover;
                update session vector for other
                    ites to indicate wait;
    else ERROR because site is not down

Receive(managing.die)
    if up or recovering then go into failure mode
    else ERROR in simulation

Receive(managing.dump)
    output information for simulation control process

Receive(managing.up)
    ERROR if unsolicited

Receive(managing.failed)
    ERROR if unsolicited

Receive(managing.xact_committed)
    ERROR , this message is sent only to the manager
```

```
Receive(managing.xact_aborted)
    ERROR , this message is sent only to the manager

Receive(managing.allow_recovery)
    if site is up then
        Send(recovery.response)  to the named site
    else ERROR

Receive(control.recovery_announce)
    if up then
        mark session vector that sender site is up;
        don't send a managing.recovery_response
            because the managing.allow_recovery message
            will inform some site to do it;
    else if in recovery wait state then
        if many sites went down together and it's ok
            to come up
            if N sites in the oldsv were marked up when
                I went down AND
                N-1 of those sites are already in
                recovery wait AND
                the control.recovery_announce is from
                the Nth site AND
                I am the highest ranked site then
                this site can come up;
                set all of the waiting site up in newsv;
                Send(control.recovery_response)  to all
                    sites;
            else if this announce is not from one of the
                sites that was up when I went down then
                    mark the sender as waiting in newsv;
                tell it to wait for sites that went down
                    later by reply Send(control.recovery_wait) ;
            else
                let it know that we went down together;
                mark the sender as waiting in newsv;
                Send(control.recovery_announce) ;
   else in down state
       reply ¡managing.failed¿;
```

```
Receive(control.recovery_response)
    if in recovery wait state then
        get new fail-locks and session vector from message;
        become operational and mark all other waiting
            sites as up;
    else ERROR

Receive(control.recovery.wait)
    if in wait state then stay there
    else ERROR

Receive(control.failure_announce)
    update session vector;
    set fail-locks for the down site for the last
        committed xact in log;
        if a transaction which is from the same source as
            the control.failure_announce is in prepare to
            commit state then
            abort that transaction;

Receive(control.clear_fail_locks)
    clear the appropriate fail-locks;

Receive(control.status)
    if in recovery wait state then
        reply Send(control.recovery_announce)
    else if in failed state then
        reply Send(managing.failed)
    else if in up state then
        reply Send(managing.up)
    else ERROR

Receive(xact.user)
    if objects in transaction have fail-locks then
        Send(xact.copier)
        Receive(xact.update)
        update database and clear fail-lock
        Send(control.clear_fail_locks)
        Receive(managing.failed)
```

```
        abort and ¡control.failure_announce¿
        if not aborted then
        Send(xact.update) to every up site
            if Receive(xact.ack) from up sites then
                update database;
                Send(xact.commit) to up sites;
                if not all Receive(xact.commit_ack) from
                    up sites then
                    Send(control.failure_announce)
                    update transaction log and set fail-locks
            else Receive(managing.failed)
                abort and ¡control.failure_announce¿

Receive(xact.update)
    if up then
        Send(xact.ack) ;
        if Receive(xact.commit) then
            update database;
            update transaction log and set fail-locks;
        else Receive(control.failure_announce) then
            see above
    else
        Send(managing.failed)

Receive(xact.ack)
    ERROR (unsolicited)

Receive(xact.commit)
    ERROR (unsolicited)

Receive(xact.commit_ack)
    ERROR (unsolicited)

Receive(xact.copier)
    if up and no fail-lock on object(s) then
        reply ¡xact.copier_update¿
    else ERROR

Receive(xact.copier_update)
    ERROR (unsolicited)
```

8 Discussion

This section considers some ideas and conclusions related to our work on the mini-RAID project.

Replication of fail-locks The full replication of fail-locks is desirable for a distributed database system in order to prevent blocking during site recovery. For example, consider a system with three sites (0, 1, and 2). Site 0 detects that site 1 has failed and informs site 2 of this fact. A user transaction is received by site 0 which contains write operations for data-items. Site 0 commits the transaction and sets fail-locks for site 1 for the data-items which were updated. Site 2 commits the transaction but does not set any fail-locks for site 1. Site 0 now fails and site 1 revives. Site 1 can receive the fail-locks which site 2 may have set for it for other user transactions, but since site 0 has failed it must block its recovery until site 0 revives in order to get the fail-locks that site 0 set.

Full replication of fail-locks requires two major actions:

1. Each site should know the status of every other site in the system and should set fail-locks as part of every transaction commit. In the example, if site 2 had set fail-locks for every committed transaction, site 2 could have sent the revived site 1 all of its fail-locks and site 1's recovery could have continued.

2. A reviving site should be sent not only the fail-locks for itself but also the fail-locks for every site in the system.

In the example, consider the case of site 2 committing other transactions while sites 0 and 1 are down, sending a reviving site 0 only the fail-locks for site 0, and then failing. Site 0 now revives but site 1 does not have the fail-locks for site 0. Site 0 must now block its recovery until site 2 revives.

The full replication of fail-locks has special implications for partially replicated databases of data-items. It suggests that each site should be informed of every committed transaction, even if a transaction has no updates for the data-items of a particular site. This could be implemented by having all sites take part in a commit protocol such that a site always votes to commit a transaction which has no updates for the site. An alternative is to have the sites which are participating in the commit protocol inform nonparticipating sites of transaction commitment.

Site Failure Given the fact that fail-locks should be fully replicated, a major problem in the area of site failure is the termination of a transaction for which a site failure (or failures)

is detected during processing of the transaction. The termination must be done such that identical fail-locks are maintained at each site which remains operational. Three possible solutions are now discussed.

1. Abort the transaction in progress, inform the other sites of the site failure(s), and restart the transaction. This allows the other sites to set fail-locks for the failed sites when the transaction is committed.

2. Implement a round of commitments similar to Skeen's termination protocol [4]. A transaction is committed and fail-locks are set only after two successive rounds of commitment with no site failure. If a failure is detected during a round, then all operational sites should be informed of the failure. The two successive rounds of non-failures insures that all operational sites are in agreement on which sites are failed and thus the fail-locks which are set for the failed sites remains consistent on all operational sites.

3. The operational sites finish commitment of the transaction in progress and then inform each other about detected site failures. The operational sites then set fail-locks for the last committed transaction.

Site Recovery Two major actions exist for a failed site which has revived: the site must announce that it is recovering to all other sites in the system and the site must receive the session numbers and fail-locks for all other sites in the system.

Announcing recovery becomes a nonproblem if the system's communications network supports message broadcasting. However, if point-to-point communications must be used, a recovering site could experience significant delays from timeouts if it tries to send to sites which are failed. If some operational sites have received the recovery announcements and are ready to begin processing with the recovering site while it is being delayed in the announcement stage, a window of vulnerability is created. Some solutions to this problem are to have a well known network address from which a site can get the system status or to have the recovering site poll the other sites one at a time to find out the system status before any announcements are sent. An operational site would respond to a poll by sending its session number vector to the recovering site. This would give the recovering site the ability to send recovery announcements to those sites which are operational and so the time spent in the announcement stage is minimized.

In receiving the fail-locks and session numbers for all other sites, it would be desirable from an efficiency point of view to have the information sent by only one operational site because the information is replicated on every site. To achieve this goal, a special flag could be included in the recovery announcement to indicate which operational site should send the necessary information.

Another interesting issue in site recovery is for the case of all operational sites failing. This requires recovering sites to determine if the last site to fail has revived. This determination becomes non-trivial if the last operational sites failed simultaneously. The section of this report on protocols outlines the strategy used by mini-RAID.

Windows of Vulnerability A window of vulnerability occurs whenever there is a difference between the actual state of a site and the state as perceived by the other sites in the system. Consider the window which occurs during site recovery. This window can be broken into two stages: the time between the sending of the recovery announcement and the reception of the recovery response (i.e., the fail-locks and session number vector for the system); and the time between the reception of the recovery response and the completion of any necessary initialization of the fail-locks and session number vector.

During the first stage, communication delays could cause other types of messages (including updates) to arrive before a recovery response message. This problem could be avoided by having a two-phase announcement protocol, but this requires the operational sites to provide special handling for any other messages which are generated during the announcement protocol. An alternative is to have the recovering site entirely handle the problem. The following is one possible solution (let *Sr* be the recovering site and *Sj* be any other operational site)::

1. *Sr* reboots, sets its own state to *recovery phase 1*, and sends the recovery announcements.

2. *Sj* receives the recovery announcement, marks *Sr* as state *up* and sends the recovery response with session-vector/fail-locks.

3. While *Sr* is in state *recovery phase 1* the following can happen:

```
if update xact received for commitment then
     update the database;
     add the xact to a list of xact to be processed;
else if copier xact received then
     send the specified data-items to the requesting
          site because no other site would request
          items for which this site has fail-locks;
else if user xact received then
     abort it or put it in a list which contains
          only the user xacts to be started after
          phase 2 of recovery;
else if any other xact then
     add the xact to a list of xact to be processed;
```

4. A recovery response arrives with new session-vector/fail-locks. The site sets its state to *recovery phase 2*. The list of non-user xacts built during the first phase is now processed along with any other non-user xacts which arrive. After processing the list of non-user xacts the site sets its state to *up* and begins processing the list of user xacts built during the recovery phases along with normal processing of any other xact.

9 Software Description

This section briefly describes the software generated for this project. Each file is listed along with the contents of the file.

comm.c This file contains routines for interprocess communications:

- Setup (wSID)
- Send (buf , SID)
- Receive (pbuf , pSID)
- ExitSite()

comm.h This header file contains data definitions for communication routines.

comutil.c This file contains communication utilities:

- send_message (message)
- receive_message (message)
- message_translation (type , subtype)

ctlutil.c This file contains utilities for use with control data such as session vectors:

- initialize_ctl_info ()
- ctl_info_io (function , site)
- get_site_state (site)
- get_old_site_state (site)
- set_site_state (site , new_state)
- set_site_snum (site , new_snum)
- announce_site_failure ()
- send_recovery_response(destination)

- send_recovery_wait(destination)
- send_recovery_announce(destination)
- send_control_status(destination)

dbsite.c This file contains the main program and initialization routine for a database site:

- main (argc , argv)
- initialize_site (site_c , num_sites_c)

dbutil.c This file contains utilities for manipulating database related data structures, including fail-locks and the database itself:

- initialize_database ()
- commit_db_values (upd_msg)
- data_item_mgr (function , arg_item , arg_value)
- fail_locks_mgr (function , site , item)
- count_fail_locks (site)

extdata.h This header file contains commonly used external data declarations.

mgr.c This file contains the main program and initialization routine for the simulation manager:

- main ()
- initialize_manager ()

mgr.h This header file contains data definitions needed by the simulation manager.

mgrcmd.c This file contains the routines to execute simulation manager commands:

- execute_mgr_command (command , site , subject_site)
- update_site_info ()
- get_failure_time (message)
- process_xact_user_msg (message , dest_site)
- build_xact_user_msg (message)
- get_random_up_site ()

mgrutil.c This file contains utility routines for the simulation manager:

- send_managing_failed(destination)

- send_managing_up(destination)
- send_managing_allow_recovery(destination,subject)

simtypes.h This header file contains all major data definitions for the project, including database structure, session vector and fail-locks structure, and message formats.

slog.c This file contains routines for logging site information:

- log_site_info (log_function , comment)
- log_message (message)

syscomm.h This header file contains data definitions for communication routines.

time.c This file contains routines used in the measurement of code execution rates:

- start_timing (comment)
- stop_timing ()

typec.c This file contains the routines to process messages of type control:

- process_control_msg (message)
- process_recovery_announce (message)
- process_recovery_response(message)
- process_recovery_wait(message)
- process_failure_announce(message)
- process_clear_fail_locks(message)
- process_status(message)

typem.c This file contains the routines to process messages of type managing:

- process_managing_msg (message)
- process_managing_revive(message)

typex.c This file contains the routines to process message of type xact:

- process_xact_msg (message)
- process_copier_xact (msg)
- process_update_xact (upd_msg)
- process_user_xact (user_msg)
- insure_current_database (user_msg , clear_msg , num_copier)

10 Using Mini-RAID

Here is an example of using the mini-RAID system. Note that each data item is given an initial value of 999. Input from the user is shown in bold type. Additional comments are shown in italics.

Run mini-RAID through the manager program.
% **mgr**

Enter the number of operations for a user xact [1-25]: **5**

Enter the number of data-items for simulation [1-100]: **50**

Enter the number of sites to be started up [2-8]: **3**
Start-up performed for site 0
Start-up performed for site 1
Start-up performed for site 2

Simulation commands:
h = help
f = fail site
r = recover site
x = send user transaction
m = send multiple user transactions to random sites
g = send mult. user xacts until fail-locks cleared
d = cause site to dump information
o = output current information
u = output information summary
c = check on children
a = send allow recovery
s = stop simulation

>>> **o**
Sat Dec 5 12:37:50 1987 : current information
actual session number: 0
session vector (site|state|session num):
0|U|01 1|U|01 2|U|01
database info (item|value|fail-locks):
0|999|– – – 1|999|– – – 2|999|– – – 3|999|– – – 4|999|– – –
5|999|– – – 6|999|– – – 7|999|– – – 8|999|– – – 9|999|– – –

```
10|999| - - - 11|999| - - - 12|999| - - - 13|999| - - - 14|999| - - -
15|999| - - - 16|999| - - - 17|999| - - - 18|999| - - - 19|999| - - -
20|999| - - - 21|999| - - - 22|999| - - - 23|999| - - - 24|999| - - -
25|999| - - - 26|999| - - - 27|999| - - - 28|999| - - - 29|999| - - -
30|999| - - - 31|999| - - - 32|999| - - - 33|999| - - - 34|999| - - -
35|999| - - - 36|999| - - - 37|999| - - - 38|999| - - - 39|999| - - -
40|999| - - - 41|999| - - - 42|999| - - - 43|999| - - - 44|999| - - -
45|999| - - - 46|999| - - - 47|999| - - - 48|999| - - - 49|999| - - -
50|999| - - - 51|999| - - - 52|999| - - - 53|999| - - - 54|999| - - -
55|999| - - - 56|999| - - - 57|999| - - - 58|999| - - - 59|999| - - -
60|999| - - - 61|999| - - - 62|999| - - - 63|999| - - - 64|999| - - -
65|999| - - - 66|999| - - - 67|999| - - - 68|999| - - - 69|999| - - -
70|999| - - - 71|999| - - - 72|999| - - - 73|999| - - - 74|999| - - -
75|999| - - - 76|999| - - - 77|999| - - - 78|999| - - - 79|999| - - -
80|999| - - - 81|999| - - - 82|999| - - - 83|999| - - - 84|999| - - -
85|999| - - - 86|999| - - - 87|999| - - - 88|999| - - - 89|999| - - -
90|999| - - - 91|999| - - - 92|999| - - - 93|999| - - - 94|999| - - -
95|999| - - - 96|999| - - - 97|999| - - - 98|999| - - - 99|999| - - -

    Write to manager log? (y,n): n
>>> u
Simulation set-up has 3 sites, xacts with 5 operations, and 50 data-items
Sat Dec 5 12:37:54 1987 : current information
    session vector (site|state|session num):
        0|U|01 1|U|01 2|U|01
    number of fail-locks currently set:
        site 0 has 0
        site 1 has 0
        site 2 has 0
Send a transaction to site 1.
>>> x
Destination site ID [0:2]: 1

Sending user transaction #1 The number of actions = 4
    Each action has the format: action|item|[value]
        R|27|xxx R|31|xxx W|30|012 W|00|308
Sat Dec 5 12:38:08 1987 : send xact.user msg to site 1
Sat Dec 5 12:38:08 1987 : received managing.xact_committed msg from site 1
Xact #0001 , Abort #0000
Site Fail-locks Copiers Copier fail-locks
```

```
 #0      00      0000      00
 #1      00      0000      00
 #2      00      0000      00
```
Send out two transactions to random sites.
>>> m

Enter number of transactions to send: 2

Sending user transaction #2 The number of actions = 5
    Each action has the format: action|item|[value]
        R|29|xxx W|49|091 R|26|xxx R|29|xxx R|14|xxx
Sat Dec 5 12:38:22 1987 : send xact.user msg to site 2
Sat Dec 5 12:38:22 1987 : received managing.xact_committed msg from site 2
Xact #0002 , Abort #0000
Site Fail-locks Copiers Copier fail-locks
 #0   00   0000   00
 #1   00   0000   00
 #2   00   0000   00

Sending user transaction #3 The number of actions = 2
    Each action has the format: action|item|[value]
        W|25|039 W|40|444
Sat Dec 5 12:38:22 1987 : send xact.user msg to site 1
Sat Dec 5 12:38:23 1987 : received managing.xact_committed msg from site 1
Xact #0003 , Abort #0000
Site Fail-locks Copiers Copier fail-locks
 #0   00   0000   00
 #1   00   0000   00
 #2   00   0000   00
Tell site 1 to fail now.
>>> f
    Destination site ID [0:2]: 1
    Failure schedule (enter H for help): N
Sat Dec 5 12:38:55 1987 : send managing.die msg to site 1
Note that the state of site 1 is now "D" for down.
>>> u
Simulation set-up has 3 sites, xacts with 5 operations, and 50 data-items
Sat Dec 5 12:39:11 1987 : current information
    session vector (site|state|session num):
        0|U|01 1|D|01 2|U|01

number of fail-locks currently set:
site 0 has 0
site 1 has 0
site 2 has 0
Send out 3 transactions, note that fail-locks are set for site 1.
>>> m

Enter number of transactions to send: **3**

Sending user transaction #4 The number of actions = 4
Each action has the format: action|item|[value]
R|19|xxx W|28|481 R|28|xxx W|13|460
Sat Dec 5 12:39:17 1987 : send xact.user msg to site 2
Sat Dec 5 12:39:17 1987 : received managing.xact_aborted msg from site 2
Xact #0004 , Abort #0001
Site Fail-locks Copiers Copier fail-locks
#0 00 0000 00
#1 00 0000 00
#2 00 0000 00

Sending user transaction #5 The number of actions = 3
Each action has the format: action|item|[value]
W|09|333 W|35|217 R|20|xxx
Sat Dec 5 12:39:17 1987 : send xact.user msg to site 2
Sat Dec 5 12:39:18 1987 : received managing.xact_committed msg from site 2
Xact #0005 , Abort #0001
Site Fail-locks Copiers Copier fail-locks
#0 00 0000 00
#1 02 0000 00
#2 00 0000 00

Sending user transaction #6 The number of actions = 2
Each action has the format: action|item|[value]
W|06|380 W|31|420
Sat Dec 5 12:39:18 1987 : send xact.user msg to site 0
Sat Dec 5 12:39:18 1987 : received managing.xact_committed msg from site 0
Xact #0006 , Abort #0001
Site Fail-locks Copiers Copier fail-locks
#0 00 0000 00
#1 04 0000 00

#2 00 0000 00

There are fail-locks for site 1 on data items 6, 9, 91, and 35.

>>> o

Sat Dec 5 12:39:29 1987 : current information

actual session number: 0

session vector (site|state|session num):

0|U|01 1|D|01 2|U|01

database info (item|value|fail-locks):

0|308| – – – 1|999| – – – 2|999| – – – 3|999| – – – 4|999| – – –
5|999| – – – 6|380| – 1 – 7|999| – – – 8|999| – – – 9|333| – 1–
10|999| – – – 11|999| – – – 12|999| – – – 13|999| – – – 14|999| – – –
15|999| – – – 16|999| – – – 17|999| – – – 18|999| – – – 19|999| – – –
20|999| – – – 21|999| – – – 22|999| – – – 23|999| – – – 24|999| – – –
25|039| – – – 26|999| – – – 27|999| – – – 28|999| – – – 29|999| – – –
30|012| – – – 31|420| – 1 – 32|999| – – – 33|999| – – – 34|999| – – –
35|217| – 1 – 36|999| – – – 37|999| – – – 38|999| – – – 39|999| – – –
40|444| – – – 41|999| – – – 42|999| – – – 43|999| – – – 44|999| – – –
45|999| – – – 46|999| – – – 47|999| – – – 48|999| – – – 49|091| – – –
50|999| – – – 51|999| – – – 52|999| – – – 53|999| – – – 54|999| – – –
55|999| – – – 56|999| – – – 57|999| – – – 58|999| – – – 59|999| – – –
60|999| – – – 61|999| – – – 62|999| – – – 63|999| – – – 64|999| – – –
65|999| – – – 66|999| – – – 67|999| – – – 68|999| – – – 69|999| – – –
70|999| – – – 71|999| – – – 72|999| – – – 73|999| – – – 74|999| – – –
75|999| – – – 76|999| – – – 77|999| – – – 78|999| – – – 79|999| – – –
80|999| – – – 81|999| – – – 82|999| – – – 83|999| – – – 84|999| – – –
85|999| – – – 86|999| – – – 87|999| – – – 88|999| – – – 89|999| – – –
90|999| – – – 91|999| – – – 92|999| – – – 93|999| – – – 94|999| – – –
95|999| – – – 96|999| – – – 97|999| – – – 98|999| – – – 99|999| – – –

Write to manager log? (y,n): n

Tell site 1 to begin recovery.

>>> r

Destination site ID [0:2]: 1

Sat Dec 5 12:39:41 1987 : send managing.revive msg to site 1

Site 1 is in state "W", waiting for replication control information.

>>> u

Simulation set-up has 3 sites, xacts with 5 operations, and 50 data-items

Sat Dec 5 12:39:48 1987 : current information

session vector (site|state|session num):

0|U|01 1|W|01 2|U|01

number of fail-locks currently set:
site 0 has 0
site 1 has 4
site 2 has 0

Tell Site 0 to send information to site 1.

>>> a

Destination site ID [0:2]: 0
Object site ID [0:2]: 1

Sat Dec 5 12:39:56 1987 : send managing.allow_recovery msg to site 0

Site 1 received information and has gone back to "U" (Up), state.

>>> u

Simulation set-up has 3 sites, xacts with 5 operations, and 50 data-items
Sat Dec 5 12:39:58 1987 : current information
session vector (site|state|session num):
0|U|01 1|U|01 2|U|01
number of fail-locks currently set:
site 0 has 0
site 1 has 4
site 2 has 0

Send out two transactions. User transaction #13 has a write for data item 31 so that fail-lock can be cleared.

>>> m

Enter number of transactions to send: 2

Sending user transaction #7 The number of actions = 2
Each action has the format: action|item|[value]
W|21|464 R|12|xxx
Sat Dec 5 12:40:30 1987 : send xact.user msg to site 0
Sat Dec 5 12:40:30 1987 : received managing.xact_committed msg from site 0
Xact #007 , Abort #0001

Site	Fail-locks	Copiers	Copier fail-locks
#0	00	0000	00
#1	04	0000	00
#2	00	0000	00

Sending user transaction #8 The number of actions = 5
Each action has the format: action|item|[value]
R|12|xxx W|17|326 W|31|013 R|18|xxx R|38|xxx
Sat Dec 5 12:40:33 1987 : send xact.user msg to site 1

Sat Dec 5 12:40:33 1987 : received managing.xact_committed msg from site 1
Xact #008 , Abort #0001
Site Fail-locks Copiers Copier fail-locks
#0 00 0000 00
#1 03 0000 00
#2 00 0000 00
Tell the sites to quit running. Examine log. for logged information.*
>>> s
Sat Dec 5 12:40:49 1987 : send managing.stop msg to site 0
Sat Dec 5 12:40:49 1987 : send managing.stop msg to site 1
Sat Dec 5 12:40:49 1987 : send managing.stop msg to site 2

References

[1] B. Bhargava, "Transaction processing and consistency control of replicated copies during failures", *Journal of Management Information Systems*, Oct. 1987, vol. 4, no. 2.

[2] B. Bhargava and J. Riedl, "The design of an adaptable distributed system", *IEEE COMP-SAC Conference*, Oct. 1986, pp. 114-122.

[3] B. Bhargava, P. Noll, and D. Sabo, "An experimental analysis of replicated copy control during site failure and recovery", *Proceedings of the 4th International Conference on Data Engineering*, IEEE, Los Angeles, CA, Feb. 1988, (to appear).

[4] D. Skeen, "A decentralized termination protocol", *Proc. Symp. on Reliability in Distributed Software and Database Syst.*, Pittsburgh, PA, July, 1981, pp. 27-32.